CE 874 - Secure Software Systems

Control Flow Integrity

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.
Run-Time protection/enforcement

- In many instances we only have access to the binary
- How do we analyze the binary for vulnerabilities?
- How do we protect the binary from exploitation?
- This would be our topic for the next few lectures
Op request

Op response

People
Processes
Computer Operations

Files
Sockets
Computer Operations

Subject

Object

[Brumley’15]
Reference Monitor: Principles

- **Complete Mediation**: The reference monitor must always be invoked.
- **Tamper-proof**: The reference monitor cannot be changed by unauthorized subjects or objects.
- **Verifiable**: The reference monitor is small enough to thoroughly understand, test, and ultimately, verify.

---

Subject \[\rightarrow\] Reference Monitor \[\rightarrow\] Object

Op request \[\rightarrow\] Op request

Op response \[\rightarrow\] Op response

Policy

[Brumley’15]
Inlined Referenced Monitor

Policy

Subject

Reference Monitor

Op request

Op response

Object

Today’s Example:
Inlining a control flow policy into a program

[Brumley’15]
Control-Flow Integrity: Principles, Implementations, and Applications
Martin Abadi, Mihai Budiu, U´lfar Erlingsson, Jay Ligatti, CCS 2005
Control Flow Integrity

• protects against powerful adversary
  • with full control over entire data memory

• widely-applicable
  • language-neutral; requires binary only

• provably-correct & trustworthy
  • formal semantics; small verifier

• efficient
  • hmm… 0-45% in experiments; average 16%

[Brumley’15]
CFI Adversary Model

Can

- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

Can Not

- Execute Data
  - NX takes care of that
- Modify Code
  - text seg usually read-only
- Write to %ip
  - true in x86
- Overwrite registers in other contexts
  - kernel will restore regs

[Brumley’15]
CFI Overview

• Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

• Method:
  • build CFG statically, e.g., at compile time
  • instrument (rewrite) binary, e.g., at install time
    • add IDs and ID checks; maintain ID uniqueness
  • verify CFI instrumentation at load time
    • direct jump targets, presence of IDs and ID checks, ID uniqueness
  • perform ID checks at run time
    • indirect jumps have matching IDs

[Brumley’15]
Control Flow Graphs
Basic Block

control is “straight”
(no jump targets except at the beginning,
no jumps except at the end)

1. \( x = y + z \)
2. \( z = t + i \)
3. \( x = y + z \)
4. \( z = t + i \)
5. jmp 1
6. jmp 3

3 static basic blocks

1. \( x = y + z \)
2. \( z = t + i \)
3. \( x = y + z \)
4. \( z = t + i \)
5. jmp 1

1 dynamic basic block

[Brumley’15]
A static Control Flow Graph is a graph where
- each vertex $v_i$ is a basic block, and
- there is an edge $(v_i, v_j)$ if there may be a transfer of control from block $v_i$ to block $v_j$.

Historically, the scope of a “CFG” is limited to a function or procedure, i.e., intra-procedural.
Call Graph

- Nodes are functions. There is an edge \((v_i, v_j)\) if function \(v_i\) calls function \(v_j\).

```c
void orange()
{
1. red(1);
2. red(2);
3. green();
}
```

```c
void red(int x)
{
    green();
    ...
}
```

```c
void green()
{
    green();
    orange();
}
```
Super Graph

• Superimpose CFGs of all procedures over the call graph

```c
void orange() {
  1. red(1);
  2. red(2);
  3. green();
}
```

```c
void red(int x) {
 ..
}
```

```c
void green() {
  green();
  orange();
}
```

A context sensitive supergraph for orange lines 1 and 2.
Precision: Sensitive or Insensitive

• The more precise the analysis, the more accurate it reflects the “real” program behavior.
  • More precise = more time to compute
  • More precise = more space
  • Limited by soundness/completeness tradeoff

• Common Terminology in any Static Analysis:
  • Context sensitive vs. context insensitive
  • Flow sensitive vs. flow insensitive
  • Path sensitive vs. path insensitive

[Brumley’15]
Soundness

If analysis says X is true, then X is true.

True Things

Things I say

Trivially Sound: Say nothing

Completeness

If X is true, then analysis says X is true.

Things I say

True Things

Trivially complete: Say everything

Sound and Complete: Say exactly the set of true things!

[Brumley’15]
Soundness, Completeness, Precision, Recall, False Negative, False Positive, All that Jazz...

Imagine we are building a classifier.

**Ground truth:** things on the left is “in”.

**Our classifier:** things inside circle is “in”.

- **Sound** means FP is empty
- **Complete** means FN is empty

Precision = \( \frac{TP}{TP+FP} \)

Recall = \( \frac{TP}{FN+TP} \)

False Positive Rate = \( \frac{FP}{TP+FP} \)

False Negative Rate = \( \frac{FN}{FN+TN} \)

Accuracy = \( \frac{TP+TN}{\Sigma \text{everything}} \)
Context Sensitive

Whether different calling contexts are distinguished

```c
void yellow() {
    1. red(1);
    2. red(2);
    3. green();
}

void red(int x) {
    ..
}

void green() {
    green();
    yellow();
}
```

Context sensitive distinguishes 2 different calls to red(-)
Context Sensitive Example

\[a = \text{id}(4);\]
\[b = \text{id}(5);\]

\[
\text{void id(int z)}  \\
\{ \text{return z; } \}
\]

Context-Sensitive (color denotes matching call/ret)

Context sensitive can tell one call returns 4, the other 5

Context-Insensitive (note merging)

\[a = \text{id}(4);\]
\[b = \text{id}(5);\]

Context insensitive will say both calls return \{4,5\}

[Brumley’15]
Flow Sensitive

• A flow sensitive analysis considers the order (flow) of statements

• Examples:
  • Type checking is flow insensitive since a variable has a single type regardless of the order of statements
  • Detecting uninitialized variables requires flow sensitivity

\[ x = 4; \]
\[ \ldots \]
\[ x = 5; \]

Flow sensitive can distinguish values of \( x \), flow insensitive cannot
Flow Sensitive Example

1. \texttt{x = 4;}
2. ....
3. n. \texttt{x = 5;}

Flow sensitive:
\texttt{x} is the constant 4 at line 1, \texttt{x} is the constant 5 at line \texttt{n}

Flow insensitive:
\texttt{x} is not a constant

[Brumley’15]
Path Sensitive

• A path sensitive analysis maintains branch conditions along each execution path
  • Requires extreme care to make scalable
  • Subsumes flow sensitivity
Path Sensitive Example

1. if (x >= 0)
2.   y = x;
3. else
4.   y = -x;

path sensitive:
- y >= 0 at line 2,
- y > 0 at line 4

path insensitive:
y is not a constant

[Brumley’15]
Precision

Even path sensitive analysis approximates behavior due to:

- loops/recursion
- unrealizable paths

1. if \(a^n + b^n = c^n \) \&\& \( n>2 \) \&\& \( a>0 \) \&\& \( b>0 \) \&\& \( c>0 \)
2. \( x = 7; \)
3. else
4. \( x = 8; \)

Unrealizable path.
\( x \) will always be 8

[Brumley’15]
Control Flow Integrity (Analysis)
CFI Overview

- Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

- Method:
  - build CFG statically, e.g., at compile time
  - instrument (rewrite) binary, e.g., at install time
    - add IDs and ID checks; maintain ID uniqueness
  - verify CFI instrumentation at load time
    - direct jump targets, presence of IDs and ID checks, ID uniqueness
  - perform ID checks at run time
    - indirect jumps have matching IDs
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}

Two possible return sites due to context insensitivity

[Brumley’15]
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

- Insert a unique number at each destination
- Two destinations are equivalent if CFG contains edges to each from the same source

*predicated* call 17, R: transfer control to R only when R has label 17

*predicated* ret 23: transfer control to only label 23

[Brumley’15]
Verify CFI Instrumentation

- Direct jump targets (e.g. call 0x12345678)
  - are all targets valid according to CFG?
- IDs
  - is there an ID right after every entry point?
  - does any ID appear in the binary by accident?
- ID Checks
  - is there a check before every control transfer?
  - does each check respect the CFG?

easy to implement correctly => trustworthy
What about indirect jumps and ret?
ID Checks

Fig. 4. Our CFI implementation of a call through a function pointer.

<table>
<thead>
<tr>
<th>Bytes (opcodes)</th>
<th>x86 assembly code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 10 00</td>
<td>ret 10h</td>
<td>return</td>
</tr>
</tbody>
</table>

is instrumented using prefetchnta destination IDs, to become:

8B 0C 24          mov  ecx, [esp]         ; load address into register
83 C4 14          add  esp, 14h         ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA cmp [ecx+4], AABBCDDH ; compare opcodes at destination
75 13             jne  error_label      ; if not ID value, then fail
FF E1             jmp  ecx             ; jump to return address
Performance

- Size: increase 8% avg
- Time: increase 0-45%; 16% avg

Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Brumley’15
Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - buffer overflow, ret2libc, pointer subterfuge, etc.

Any check becomes non-circumventable.

- Allow data-only attacks since they respect CFG!
  - incorrect usage (e.g. printf can still dump mem)
  - substitution of data (e.g. replace file names)
Software Fault Isolation

• SFI ensures that a module only accesses memory within its region by adding checks
  • e.g., a plugin can accesses only its own memory

\[
\text{if} (\text{module}_\text{lower} < x < \text{module}_\text{upper}) \\
z = \text{load}[x];
\]

• CFI ensures inserted memory checks are executed
Inline Reference Monitors

- IRMs inline a security policy into binary to ensure security enforcement

- Any IRM can be supported by CFI + Software Memory Access Control
  - CFI: IRM code cannot be circumvented
  - SMAC: IRM state cannot be tampered
Accuracy vs. Security

- The accuracy of the CFG will reflect the level of enforcement of the security mechanism.

```cpp
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
```

Indistinguishable sites, e.g., due to lack of context sensitivity will be merged.
Context Sensitivity Problems

• Suppose A and B both call C.
• CFI uses same return label in A and B.

• How to prevent C from returning to B when it was called from A?
• Shadow Call Stack
  • a protected memory region for call stack
  • each call/ret instrumented to update shadow
  • CFI ensures instrumented checks will be run
CFI Summary

• Control Flow Integrity ensures that control flow follows a path in CFG
  • Accuracy of CFG determines level of enforcement
  • Can build other security policies on top of CFI
Code Pointer Integrity
Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Canda, R. Sekar, Dawn Song, OSDI 2014
Control-Flow Hijack Attack

① int *q = buf + input;
② *q = input2;
③ (*func_ptr)();

① Attacker corruptions a data pointer
② Attacker uses it to overwrite a code pointer
③ Control-flow is transferred to shell code

[Kuznetsov’14]
Memory safety prevents control-flow hijacks

- ... but memory safe programs still rely on C/C++ ...
- Sample Python program (Dropbox SDK example):

<table>
<thead>
<tr>
<th>Python program</th>
<th>3 KLOC of Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python runtime</td>
<td>500 KLOC of C</td>
</tr>
<tr>
<td>libc</td>
<td>2500 KLOC of C</td>
</tr>
</tbody>
</table>

[Kuznetsov’14]
Memory safety can be retrofitted to C/C++

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoftBound+CETS</td>
<td>116%</td>
</tr>
<tr>
<td>CCured (language modifications)</td>
<td>56%</td>
</tr>
<tr>
<td>Watchdog (hardware modifications)</td>
<td>29%</td>
</tr>
<tr>
<td>AddressSanitizer (approximate)</td>
<td>73%</td>
</tr>
</tbody>
</table>

[Kuznetsov’14]
State of the art: Control-Flow Integrity

Static property:
limit the set of functions that can be called at each call site

Coarse-grained CFI
and
Finest-grained CFI

can be bypassed [1-4]

has 10-21% overhead [5-6]

Programmers have to choose

Safety Security vs Flexibility Performance
Code-Pointer Integrity, provides both

- Control-flow hijack protection
- Practical protection
- Guaranteed protection

and

- Unmodified C/C++
- 0.5 - 1.9% overhead
- 8.4 - 10.5% overhead

Key insight: memory safety for code pointers only.

Tested on:

- FreeBSD hardened
- Python
- SQLite
- LAME
- GraphicsMagick
- OpenSSL
- PostgreSQL
- Apache

Spring 1398

Ce 874 - Control Flow Integrity

[Kuznetsov’14]
Threat Model

- Attacker can read/write data, read code
- Attacker cannot
  - Modify program code
  - Influence program loading
Memory Safety: program instrumentation

```
char *buf = malloc(10);
buf_lower = p; buf_upper = p+10;
...
char *q = buf + input;
q_lower = buf_lower; q_upper = buf_upper;
if (q < q_lower || q >= q_upper - size)
    abort();
*q = input2;
...
(*func_ptr)();
```

116% average performance overhead (Nagarakatte et al., PLDI’09 and ISMM’10)

All-or-nothing protection
Memory Safety

116% average performance overhead

Can memory safety be enforced for code pointers only?

Control-flow hijack protection
1.9% or 8.4% average performance overhead

[Kuznetsov’14]
Practical Protection (CPS): Heap

```
int *q = buf + input;
*q = input2;
...
(*func_ptr)();
```

Instructions that access code pointers are identified using type-based static analysis.

Separation is enforced using hardware-enforced instruction-level isolation.

Code pointers only

**Safe Memory**

- `func_ptr`
- 2.5% memory accesses (on SPEC2006 CPU)

**Regular Memory**

- `buf`
- 97.5% memory accesses (on SPEC2006 CPU)

All non-code-pointer data

Memory layout unchanged

[Kuznetsov’14]
Practical Protection (CPS): Stack

```c
int foo() {
    char buf[16];
    int r;
    r = scanf("%s", buf);
    return r;
}
```

All locals that are only accessed safely

Safe Stack

- `r`
- `ret address`

Stacks are separated

Regular Stack

- `buf`

Only locals accessed through pointers

Not needed in most small functions

Safe stack adds <0.1% performance overhead!
Practical Protection (CPS): Memory Layout

Safe memory
(code pointers)

Safe Heap

Safe Stack (thread1)

Safe Stack (thread2)

\[ \ldots \]

Regular memory
(non-code-pointer data)

Regular Heap

Regular Stack (thread1)

Regular Stack (thread2)

\[ \ldots \]

Code (Read-Only)

Only instructions that operate on code pointers can access the safe memory

Hardware-based instruction-level isolation

Kuznetsov'14
The CPS Promise

Under CPS, an attacker cannot forge a code pointer
Under CPS, an attacker cannot forge a code pointer

Contrived example of an attack on a CPS-protected program

1. int *q = p + input;
2. *q = input2;
3. func_ptr = struct_ptr->f;
4. (*func_ptr)();

① Attacker corrupts a data pointer
② Attacker uses it to corrupt a struct pointer
③ Program loads a function pointer from wrong location in the safe memory
④ Control-flow is transferred to different function whose address was previously stored in the safe memory
Under CPS, an attacker cannot forge a code pointer

Contrived example of an attack on a CPS-protected program

```c
int *q = p + input;
*q = input2;
...
func_ptr = struct_ptr->f;
(*func_ptr)();
```

Precise solution: protect all sensitive\(^1\) pointers

\(^1\)Sensitive pointers = code pointers and pointers used to access sensitive pointers

[Kuznetsov’14]
Code-Pointer Separation

- Identify Code-Pointer accesses using static type-based analysis
- Separate using instruction-level isolation (e.g., segmentation)

- CPS security guarantees
  - An attacker cannot forge new code pointers
  - Code-Pointer is either immediate or assigned from code pointer
  - An attacker can only replace existing functions through indirection: e.g., foo->bar->func() vs. foo->bar->func2()
Code-Pointer Integrity (CPI)

- Sensitive Pointers = code pointers and pointers used to access sensitive pointers

- CPI identifies all sensitive pointers using an over-approximate type-based static analysis:
  \[ \text{is_sensitive}(v) = \text{is_sensitive_type}(\text{type of } v) \]

- Over-approximation only affects performance
  - On SPEC2006 <= 6.5% accesses are sensitive
Guaranteed Protection (CPI): Memory Layout

Accesses are checked for memory safety

Safe memory
(sensitive pointers and metadata)

Safe Heap
Safe Stack (thread1) Safe Stack (thread2) ...

Only instructions that operate on sensitive pointers can access the safe memory

Hardware-based instruction-level isolation

Regular memory
(non-sensitive data)

Regular Heap
Regular Stack (thread1) Regular Stack (thread2) ...

Code (Read-Only)

Accesses are fast

[Kuznetsov’14]
Guaranteed Protection (CPI)

- Guaranteed memory safety for all sensitive pointers
  - Sensitive Pointers = code pointers and *pointers used to access sensitive pointers*

- $\Rightarrow$ Guaranteed protection against control-flow hijack attacks enabled by memory bugs
Code-Pointer Integrity vs. Separation

- Separate sensitive pointers from regular data
  - Type-based static analysis
  - Sensitive pointers = code pointers + pointers to sensitive pointers

- Accessing sensitive pointers is **safe**
  - Separation + runtime (bounds) checks

- Accessing regular data is **fast**
  - Instruction-level safe region isolation
Security Guarantees

- **Code-Pointer Integrity**: formally guaranteed protection
  - 8.4% to 10.5% overhead (~6.5% of memory accesses)
- **Code-Pointer Separation**: strong protection in practice
  - 0.5% to 1.9% overhead (~2.5% of memory accesses)
- **Safe Stack**: full ROP protection
  - Negligible overhead
<table>
<thead>
<tr>
<th>Protects Against</th>
<th>Technique</th>
<th>Security Guarantees</th>
<th>Average Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory corruption vulnerabilities</td>
<td>Memory Safety</td>
<td>Precise</td>
<td>116%</td>
</tr>
<tr>
<td></td>
<td><strong>CPI</strong> (Guaranteed protection)</td>
<td>Precise</td>
<td>8.4-10.5%</td>
</tr>
<tr>
<td></td>
<td><strong>CPS</strong> (Practical protection)</td>
<td>Strong</td>
<td>0.5-1.9%</td>
</tr>
<tr>
<td>Control-flow hijack vulnerabilities</td>
<td>Finest-grained CFI</td>
<td>Medium (attacks may exist)</td>
<td>10-21%</td>
</tr>
<tr>
<td></td>
<td>Coarse-grained CFI</td>
<td>Weak (known attacks)</td>
<td>4.2-16%</td>
</tr>
<tr>
<td></td>
<td>ASLR DEP Stack cookies</td>
<td>Weakest (bypassable + widespread attacks)</td>
<td>~0%</td>
</tr>
</tbody>
</table>

**References:**

- Göktas et al., IEEE S&P 2014
- Göktaş el., USENIX Security 2014
- Davi et al., USENIX Security 2014
- Carlini et al., USENIX Security 2014

**Notes:**

- Precise security guarantees mean that the system can prevent all known attacks.
- Strong security guarantees mean that the system can prevent most known attacks.
- Medium security guarantees mean that the system can prevent some known attacks.
- Weak security guarantees mean that the system can prevent some known attacks but not all.
- Weakest security guarantees mean that the system can prevent neither known attacks nor widespread attacks.

**Average Overhead:**

- CPI: 8.4-10.5%
- CPS: 0.5-1.9%
- Finest-grained CFI: 10-21%
- Coarse-grained CFI: 4.2-16%
- ASLR DEP Stack cookies: ~0%

- [Kuznetsov’14]
Implementation

• LLVM-based prototype
  • Front end (clang): collect type information
  • Back-end (llvm): CPI/CPS/SafeStack instrumentation pass
  • Runtime support: safe heap and stack management
  • Supported ISA's: x64 and x86 (partial)
  • Supported systems: Mac OSX, FreeBSD, Linux
Current status

- Great support for CPI on Mac OSX and FreeBSD on x64
- Upstreaming in progress
  - Safe Stack coming to LLVM soon
  - Fork it on GitHub now: https://github.com/cpi-llvm
- Code-review of CPS/CPI in process
  - Play with the prototype: http://levee.epfl.ch/levee-early-preview-0.2.tgz
  - Will release more packages soon
- Some changes to super complex build systems needed
  - Adapt Makefiles for FreeBSD
Conclusion

• CPI/CPS offers strong control-flow hijack protection
  • Key insight: memory safety for code pointers only

• Working prototype
  • Supports unmodified C/C++, low overhead in practice
  • Upstreaming patches in progress, SafeStack available soon!

• Homepage: http://levee.epfl.ch
• GitHub: https://github.com/cpi-llvm
Acknowledgments/References

• [Brumley’15] Introduction to Computer Security (18487/15487), David Brumley and Vyas Sekar, CMU, Fall 2015.
